



Vector Processing


(aka, Single Instruction Multiple Data, or SIMD)



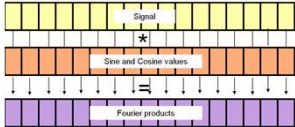
Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



Oregon State University
Computer Graphics



mjb - April 26, 2023

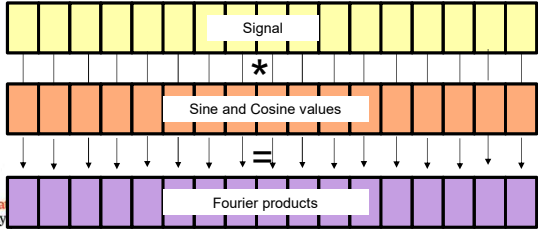
What is Vectorization/SIMD and Why do We Care?

Performance!

Many hardware architectures today, both CPU and GPU, allow you to perform arithmetic operations on multiple array elements simultaneously.

(Thus the label, "Single Instruction Multiple Data".)

We care about this because many problems, especially scientific and engineering, can be cast this way. Examples include convolution, Fourier transform, power spectrum, autocorrelation, etc.



mjb - April 26, 2023

SIMD in Intel Chips

Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

←

Xeon Phi

←

Note: one complete cache line!
Also note: a 4x4 transformation matrix!

If you care:

- MMX stands for "MultiMedia Extensions"
- SSE stands for "Streaming SIMD Extensions"
- AVX stands for "Advanced Vector Extensions"

mjb - April 26, 2023

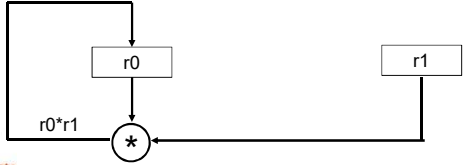
Intel SSE

Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

Intel and AMD CPU architectures support vectorization. The most well-known form is called Streaming SIMD Extension, or **SSE**. It allows four floating point operations to happen simultaneously.

Normally a *scalar* floating point multiplication instruction happens like this:

mulss r1, r0 ← "ATT form": mulss src, dst



mjb - April 26, 2023

Intel SSE

The SSE version of the multiplication instruction happens like this:

`mulps xmm1, xmm0` ← "ATT form":
mulps src, dst

Oregon State University
Computer Graphics

mjb - April 26, 2023

SSE in the Kitchen? ☺

`mulss r1, r0` `mulps xmm1, xmm0`

Oregon State University
Computer Graphics

mjb - April 26, 2023

SIMD using OpenMP SIMD Pragma

Array * Array

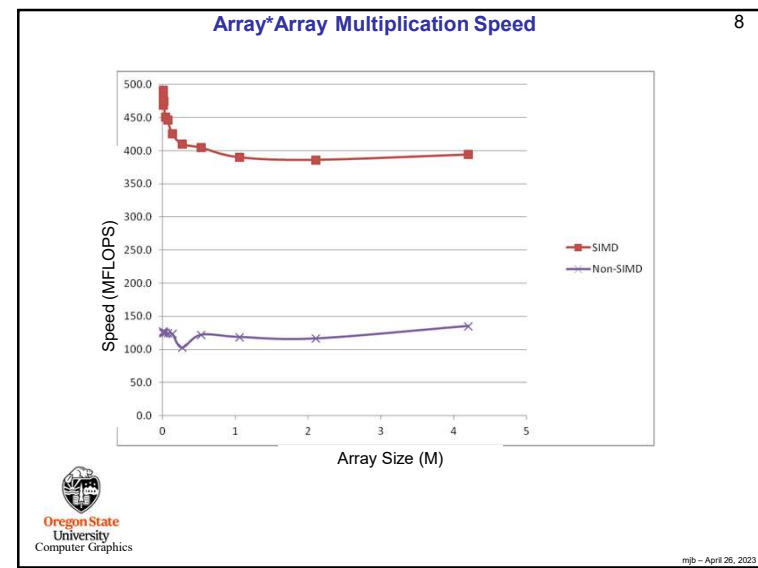
```
void
SimdMul( float *a, float *b, float *c, int len )
{
    #pragma omp simd
    for( int i = 0; i < len; i++ )
        c[i] = a[i] * b[i];
}
```

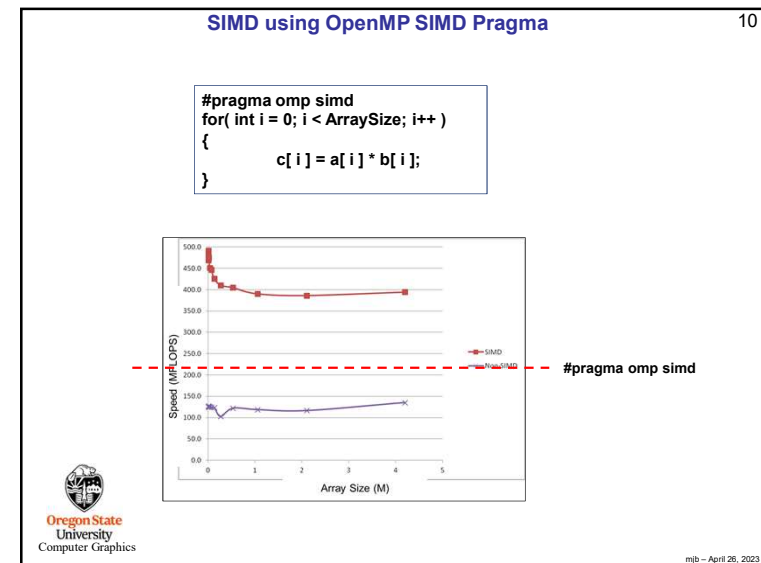
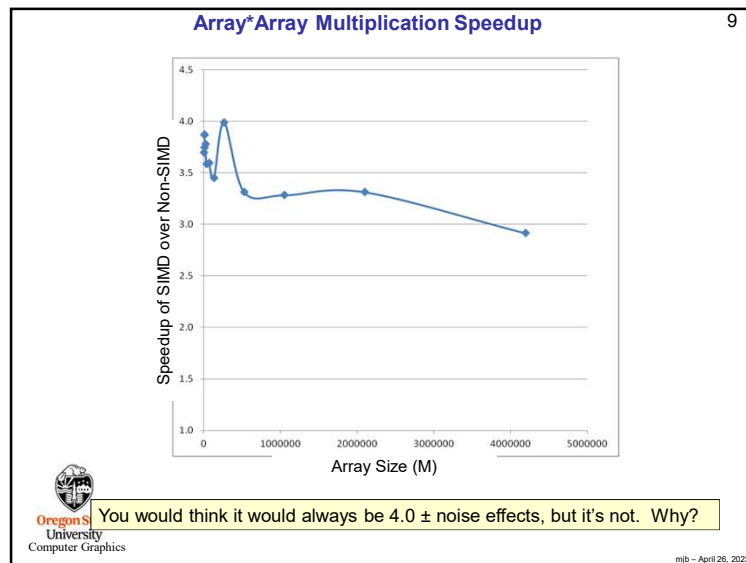
Array * Scalar

```
void
SimdMul( float *a, float b, float *c, int len )
{
    #pragma omp simd
    for( int i = 0; i < len; i++ )
        c[i] = a[i] * b;
}
```

Oregon State University
Computer Graphics

mjb - April 26, 2023





Requirements for a For-Loop to be Vectorized

- If there are nested loops, the one to vectorize must be the inner one.
- There can be no jumps or branches. "Masked assignments" (an if-statement-controlled assignment) are OK, e.g.,


```
if( A[ i ] > 0. )
    B[ i ] = 1.;
```
- The total number of iterations must be known at runtime when the loop starts
- There can be no inter-loop data dependencies such as:


```
a[ i ] = a[ i-1 ] + 1.;
```

101st element 100th element

a[100] = a[99] + 1.; // this crosses an SSE boundary, so it is OK

a[101] = a[100] + 1.; // this is within one SSE operation, so it is NOT OK

102nd element 101st element

- It helps performance if the elements have contiguous memory addresses.

Oregon State University Computer Graphics

mjb - April 26, 2023

Prefetching

Prefetching is used to place a cache line in memory before it is to be used, thus hiding the latency of fetching from off-chip memory.

There are two key issues here:

- Issuing the prefetch at the right time
- Issuing the prefetch at the right distance

The right time:
If the prefetch is issued too late, then the memory values won't be back when the program wants to use them, and the processor has to wait anyway.

If the prefetch is issued too early, then there is a chance that the prefetched values could be evicted from cache by another need before they can be used.

The right distance:
The "prefetch distance" is how far ahead the prefetch memory is than the memory we are using right now.

Too far, and the values sit in cache for too long, and possibly get evicted.

Too near, and the program is ready for the values before they have arrived.

Oregon State University Computer Graphics

mjb - April 26, 2023

The Effects of Prefetching on SIMD Computations

13

Array Multiplication

Length of Arrays (NUM): 1,000,000
Length per SIMD call (ONETIME): 256

```
for( int i = 0; i < NUM; i += ONETIME )
{
    __builtin_prefetch ( &A[i+PD], WILL_READ_ONLY, LOCALITY_LOW );
    __builtin_prefetch ( &B[i+PD], WILL_READ_ONLY, LOCALITY_LOW );
    __builtin_prefetch ( &C[i+PD], WILL_READ_AND_WRITE, LOCALITY_LOW );

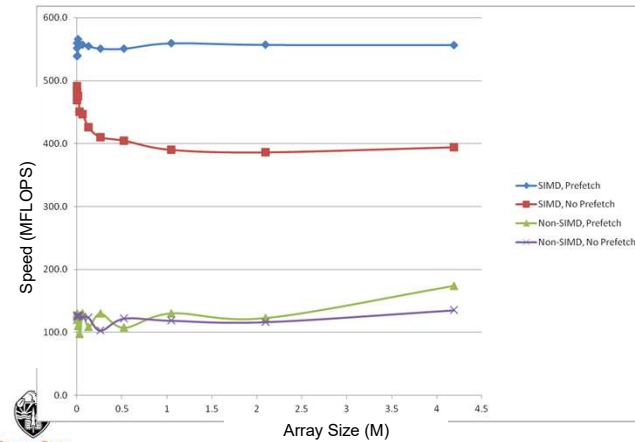
    SimdMul( A, B, C, ONETIME );
}
```



mjb - April 26, 2023

The Effects of Prefetching on SIMD Computations

14



mjb - April 26, 2023

This all sounds great! What is the catch?

15

The catch is that compilers haven't caught up to producing really efficient SIMD code. So, while there are great ways to express the desire for SIMD in code, you won't get the full potential speedup ... yet.

One way to get a better speedup is to use assembly language. Don't worry – you wouldn't need to write it.

Here are two assembly functions:

1. SimdMul: $C[0:len] = A[0:len] * B[0:len]$
2. SimdMulSum: $\text{return} (\sum A[0:len] * B[0:len])$

Warning – due to the nature of how different compilers and systems handle local variables, these two functions only work on *flap* using gcc/g++, without -O3 !!!



mjb - April 26, 2023

Getting at the full SIMD power until compilers catch up

16

```
void
SimdMul( float *a, float *b, float *c, int len )
{
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
    __asm
    (
        ".att_syntax\n\t"
        "movq -24(%rbp), %r8\n\t" // a
        "movq -32(%rbp), %rcx\n\t" // b
        "movq -40(%rbp), %rdx\n\t" // c
    );

    for( int i = 0; i < limit; i += SSE_WIDTH )
    {
        __asm
        (
            ".att_syntax\n\t"
            "movups (%r8), %xmm0\n\t" // load the first sse register
            "movups (%rcx), %xmm1\n\t" // load the second sse register
            "mulps %xmm1, %xmm0\n\t" // do the multiply
            "movups %xmm0, (%rdx)\n\t" // store the result
            "addq $16, %r8\n\t"
            "addq $16, %rcx\n\t"
            "addq $16, %rdx\n\t"
        );
    }

    for( int i = limit; i < len; i++ )
    {
        c[i] = a[i] * b[i];
    }
}
```

This only works on *flap* using gcc/g++, without -O3 !!!



mjb - April 26, 2023

Getting at the full SIMD power until compilers catch up

17

```
float
SimdMulSum( float *a, float *b, int len )
{
    float sum[4] = { 0., 0., 0., 0. };
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;

    {
        __asm
        {
            ".att_syntax"
            "movq -40(%rbp), %r8" // a
            "movq -48(%rbp), %r9" // b
            "leaq -32(%rbp), %rdx" // &sum[0]
            "movups (%rdx), %xmm2" // 4 copies of 0. in xmm2
        };

        for( int i = 0; i < limit; i += SSE_WIDTH )
        {
            __asm
            {
                ".att_syntax"
                "movups (%r8), %xmm0" // load the first sse register
                "movups (%r9), %xmm1" // load the second sse register
                "mulps %xmm1, %xmm0" // do the multiply
                "addps %xmm0, %xmm2" // do the add
                "addq $16, %r8"
                "addq $16, %r9"
            };
        };

        __asm
        {
            ".att_syntax"
            "movups %xmm2, (%rdx)" // copy the sums back to sum[ ]
        };

        for( int i = limit; i < len; i++ )
        {
            sum[0] += a[i] * b[i];
        };

        return sum[0] + sum[1] + sum[2] + sum[3];
    }
}
```

This only works on *flip* using gcc/g++,
without -O3 !!!



mjb - April 26, 2023

Each Core Has Its Own SIMD Unit! Thus, You Should be able to Combine Multicore and SIMD

18

```
#define NUM_ELEMENTS_PER_CORE    ( ARRAYSIZE / NUMT )
```

```
...
```

```
omp_set_num_threads( NUMT );
double maxMegaMultsPerSecond = 0.;
```

```
double time0 = omp_get_wtime( );
```

```
#pragma omp parallel
```

```
{
    int thisThread = omp_get_thread_num( );
    int first = thisThread * NUM_ELEMENTS_PER_CORE;
    SimdMul( &A[first], &B[first], &C[first], NUM_ELEMENTS_PER_CORE );
}
```

```
double time1 = omp_get_wtime( );
double megaMultsPerSecond = (double)ARRAYSIZE / ( time1 - time0 ) / 1000000.;
```

```
...
```

The variable *first* is the first array element that *thisThread* is in charge of.

&A[first] is the memory address of that thread's first element.



mjb - April 26, 2023

Combining SIMD with Multicore

19

Notes:

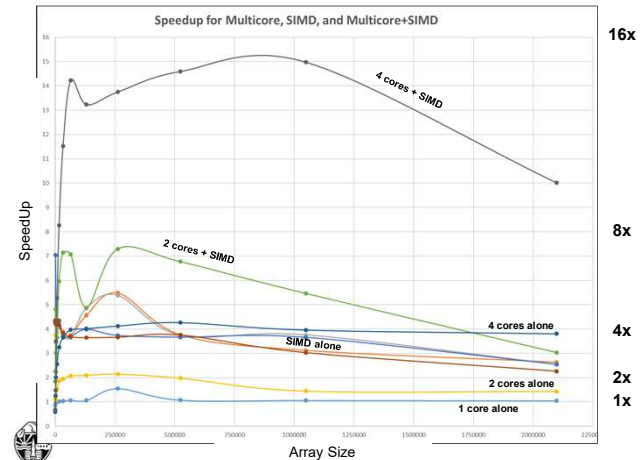
- Remember that **#pragma omp parallel** creates a thread team and that *all* threads execute *everything* in the curly braces.
- The variable **thisThread** is the thread number of the thread who is executing this code right now. There will eventually be NUMT threads who get to execute this code. Thus, all the instances of **thisThread** will be between 0 and NUMT-1.
- The variable **first** is the first array element number that **thisThread** will execute.
- Starting the SIMD multiplications at **&A[first]**, **&B[first]**, **&C[first]** gives each thread its very own set of contiguous array elements to work on. The **SimdMul** function depends on this.



mjb - April 26, 2023

Combining SIMD with Multicore

20



- Speedups are with respect to a for-loop with no multicore or SIMD.
- "cores alone" = a for-loop with "#pragma omp parallel for".
- "cores + SIMD" = as the code looks on last two slides



mjb - April 26, 2023

Avoiding Assembly Language: the Intel Intrinsics

21

Intel has a mechanism to get at the SSE SIMD without resorting to assembly language. These are called **Intrinsics**.

Intrinsic	Meaning
<code>__m128</code>	Declaration for a 128 bit 4-float word
<code>_mm_loadu_ps</code>	Load a <code>__m128</code> word from memory
<code>_mm_storeu_ps</code>	Store a <code>__m128</code> word into memory
<code>_mm_mul_ps</code>	Multiply two <code>__m128</code> words
<code>_mm_add_ps</code>	Add two <code>__m128</code> words



mjb - April 26, 2023

SimdMul using Intel Intrinsics

22

```
#include <xmmintrin.h>
#define SSE_WIDTH 4

void
SimdMul( float *a, float *b, float *c, int len )
{
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
    register float *pa = a;
    register float *pb = b;
    register float *pc = c;
    for( int i = 0; i < limit; i += SSE_WIDTH )
    {
        _mm_storeu_ps( pc, _mm_mul_ps( _mm_loadu_ps( pa ), _mm_loadu_ps( pb ) ) );
        pa += SSE_WIDTH;
        pb += SSE_WIDTH;
        pc += SSE_WIDTH;
    }

    for( int i = limit; i < len; i++ )
    {
        c[i] = a[i] * b[i];
    }
}
```



mjb - April 26, 2023

SimdMulSum using Intel Intrinsics

23

```
float
SimdMulSum( float *a, float *b, int len )
{
    float sum[4] = { 0., 0., 0., 0. };
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
    register float *pa = a;
    register float *pb = b;

    __m128 ss = _mm_loadu_ps( &sum[0] );
    for( int i = 0; i < limit; i += SSE_WIDTH )
    {
        ss = _mm_add_ps( ss, _mm_mul_ps( _mm_loadu_ps( pa ), _mm_loadu_ps( pb ) ) );
        pa += SSE_WIDTH;
        pb += SSE_WIDTH;
    }
    _mm_storeu_ps( &sum[0], ss );

    for( int i = limit; i < len; i++ )
    {
        sum[0] += a[i] * b[i];
    }

    return sum[0] + sum[1] + sum[2] + sum[3];
}
```



mjb - April 26, 2023

Intel Intrinsics

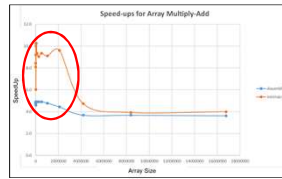
24



mjb - April 26, 2023

Why do the Intrinsics do so well with a small dataset size?

25



It's not due to the code in the inner-loop:

C/C++	Assembly	Intrinsics
<pre>for(int i = 0; i < len; i++) { c[i] = a[i] * b[i]; }</pre>	<pre>movups (%r8), %xmm0 movups (%rcx), %xmm1 mulps %xmm1, %xmm0 movups %xmm0, (%rdx) addq \$16, %r8 addq \$16, %rcx addq \$16, %rdx addl \$4, -4(%rbp)</pre>	<pre>movups (%r10), %xmm0 movups (%r9), %xmm1 mulps %xmm1, %xmm0 movups %xmm0, (%r11) addq \$16, %r9 addq \$16, %r10 addq \$16, %r11 addl \$4, %r8d</pre>

It's actually due to the setup time. The intrinsics have a tighter coupling to the setting up of the registers. A smaller setup time makes the small dataset size speedup look better.

A preview of things to come: OpenCL and CUDA have SIMD Data Types

26

When we get to OpenCL, we could compute projectile physics like this:

```
float4 pp; // p'
pp.x = p.x + v.x*DT;
pp.y = p.y + v.y*DT + .5*DT*DT*G.y;
pp.z = p.z + v.z*DT;
```

But, instead, we will do it like this:

```
float4 pp = p + v*DT + .5*DT*DT*G; // p'
```

We do it this way for two reasons:

1. Convenience and clean coding
2. Some hardware can do multiple arithmetic operations simultaneously

A preview of things to come: OpenCL and CUDA have SIMD Data Types

27

The whole thing will look like this:

```
constant float4 G = (float4) ( 0., -9.8, 0., 0. );
constant float DT = 0.1;

kernel
void
Particle( global float4 * dPobj, global float4 * dVel, global float4 * dCobj )
{
    int gid = get_global_id( 0 ); // particle #
    float4 p = dPobj[gid]; // particle #gid's position
    float4 v = dVel[gid]; // particle #gid's velocity

    float4 pp = p + v*DT + .5*DT*DT*G; // p'
    float4 vp = v + G*DT; // v'

    dPobj[gid] = pp;
    dVel[gid] = vp;
}
```



28

- SIMD is an important way to achieve speed-ups on a CPU
- For now, you might have to write in assembly language or use Intel intrinsics to get to all of it
- I suspect that `#pragma omp simd` will eventually catch up
- Prefetching can really help SIMD